Object-Oriented Programming 22

Object-oriented programming is not hard to learn. In fact, the hardest thing about object-oriented programming is unlearning what you already know about procedural (traditional) programming. To make object-oriented programming easier to learn, this chapter uses examples and comparisons to procedural programming. The examples should make some concepts more concrete, and the comparisons to procedural programming will help you relate what you're learning to something you already know.

Contents

Contents 415
Overview 417

Objects and Messages 417

Classes 418

Inheritance and Polymorphism 419

Objects and the Macintosh Interface 420

Working with Objects 421
- What classes should I define? 421
- When should I create a subclass? 421
- What should be a method? 421
- When should I use procedural programming? 422

Where to Go Next 422

22 Object-Oriented Programming

Overview

The basic difference between procedural and object-oriented programming is in the way the two disciplines treat data and action. In procedural programming, data and action are two separate things. You define your data structures, and then you define some routines to operate on them. For each data structure you define, you need a new set of routines.

In object-oriented programming, action and data are closely coupled. When you define your data—your objects—you also define their actions. Instead of a set of routines that do something to data, you have a set of objects interacting with each other.

Objects and Messages

An object is an entity that contains some data and an associated set of actions that operate on the data. To make an object perform one of its actions, you send it a message.

For example, you might create an object that represents a rectangle. Its data contains the locations of the rectangle's four corner points, and its actions might include drawing, erasing, and moving. To draw a rectangle, you send the rectangle a draw message.

Note

For the time being, don't worry about *how* to send a message to an object. The important thing is to start thinking of sending messages as requesting an object to do something.

Contrast this with the way you'd do the same thing in procedural programming. First you define a record that represents the four corners of the data structures, and then you define three routines: one to draw, one to erase, and one to move a rectangle. Each of the routines would take a rectangle data structure as an argument.

So one advantage of object-oriented programming over procedural programming is that you can keep the routines that operate on a data structure together with the data structure they're operating on. This "keeping together" is called encapsulation.

Classes

Encapsulation is a minor advantage of object-oriented programming. A more significant advantage is that objects can inherit data and behavior from other objects.

Every object belongs to a class, which defines the implementation of a particular kind of object. A class describes the object's data and the messages it responds to.

Classes are very much like record declarations. You define the private data for the class the same way you would the fields of a record. In classes, though, the fields are called instance variables. Each instance of a class—each object—has its own instance variables, just as each variable of a record type has the same fields.

When you send a message to an object , it invokes a routine that implements that message. These routines are called methods. The class definition includes the method implementations.

One important thing to keep in mind is that message and method are not the same thing. A message is what you send to an object. How an object responds to a message is the method.

You can think of a class as a template for creating objects. It describes an object's data and the messages it responds to. An object is called an instance of a class. You can also say that an object is a member of a class.

The rest of this chapter uses pictures like this to represent classes. This is what the rectangle class in the example on the previous page looks like:

Class	`Rectangle`
Instance variables	

`top`
`left`
`bottom`
`right`	

Messages	**Methods**
`Draw`	`draw line from point to point`
`Erase`	`erase lines`
`Move`	`offset points`

Inheritance and Polymorphism

You can define a class in terms of an existing class. The new class is called the subclass, and the existing class is called the superclass. A class without a superclass is said to be a root class. A subclass inherits all the instance variables and methods of its superclass. Subclasses can define additional instance variables and methods. Subclasses can also override methods defined by the superclass.

Overriding a method means that the subclass responds to the same message as its superclass, but it uses its own method to respond to the message.

For example, suppose you want to create a class to represent employees. Two of the instance variables might be the person's name and birth date. You might define three methods—`GetName`, `GetAge`, and `GetWkPay`—to return the person's name, to calculate the person's age from the birth date, and to return the person's weekly salary.

Class	`Employee`
Superclass	`none`
Instance variables	
`Name` `Birthdate`	
Messages	**Methods**
`GetName`	`return Name`
`GetAge`	`return (Today - Birthdate)`
`GetWkPay`	`return 0`

Now you can use the `Employee` class to create two new subclasses: `HourlyEmployee` and `ExemptEmployee`. Both of these classes inherit all the instance variables from the `Employee` class. Each class, however, defines a new instance variable to store the salary, and each class overrides the `GetWkPay` method to return the appropriate weekly salary.

This is what the HourlyEmployee class looks like:

Class	HourlyEmployee
Superclass	Employee
Instance variables	
HourlySalary	
Messages	**Methods**
GetWkPay	return (HourlySalary * 40)

And this is what the ExemptEmployee class looks like:

Class	ExemptEmployee
Superclass	Employee
Instance variables	
YearlySalary	
Messages	**Methods**
GetWkPay	return (YearlySalary / 52)

As long as your object is a member of any of the Employee classes, you can send it GetName, GetAge, and GetWkPay messages and the object will respond properly.

This ability to send the same message to objects of different classes is called polymorphism. If you need to create a new kind of employee (a salesperson on commission), all you need to do is define a subclass of Employee and override the GetWkPay method. Any routine that sends GetWkPay messages to employee objects would still work.

Consider for a moment how you would do the same thing in procedural programming. Your employee record would need a flag field to specify whether it was an hourly employee or an exempt employee. Then, your GetWkPay function would check this field and calculate the weekly salary appropriately. If you added a salesperson, you would have to change the GetWkPay function to handle the salary calculation for the new kind of employee.

Objects and the Macintosh Interface

The Macintosh interface lends itself to object-oriented programming. For instance, in the Finder, the effect of the **Open** command depends on the icons you select. If you select a folder, it opens the folder. If you select an application, it launches the application. If you

select a document, it launches the application and tells it to open the selected document. What you're doing is sending an `Open` message to different Finder objects. Each object uses its own `Open` method to carry out the request.

Think of the visual entities on the Macintosh screen—the windows, the controls, the menu bar, the icons—as objects. They're things that are waiting to do something. Think of your actions—clicking, dragging, typing—as messages. You're sending messages to the Macintosh objects.

Working with Objects

If you've never worked with object-oriented programming, this section will give you some general guidelines for designing classes. As you get more comfortable with object-oriented programming, you'll build up a collection of classes you can use in any of your applications.

What classes should I define?

Usually, you define only one root-level class and then define all the other classes as descendants of that class. The advantage of this approach is that you can define behavior that applies to all the objects in your application.

This root-level class is an example of an abstract class. You don't create instances of abstract classes. Instead, you use them to give a family of objects common behavior. The `Employee` class in the example above is an abstract class. Its function is to provide the common instance variables and methods you'll need for all employees.

In general, you should define a class (or a subclass of your root class) for each concept your application deals with.

When should I create a subclass?

Create a subclass whenever you need to change the behavior of a method or when you need to add more instance variables. If you write a method that does different things depending on the value of some instance variable, it's probably time to create a subclass.

What should be a method?

Typically, any action an object can make should be a method. It's considered good practice to provide methods to set and get the

values of instance variables rather than allowing other functions to access them directly. If you find yourself passing an object as a parameter to a function to manipulate its instance variables, you should turn the function into a method.

When should I use procedural programming?

Object-oriented programming isn't the answer to all programming problems. Some problems are solved best by old-fashioned procedural programming. In particular, procedural programming is better for procedures that are highly algorithmic or computationally intensive.

Where to Go Next

The next chapter describes how THINK C implements object-oriented programming. THINK C uses the syntax and many of the concepts defined in the C++ language. The chapter is fairly long and complex.

If you're not familiar with object-oriented programming, don't read all of Chapter 24. Instead, read up to section "Overview" on page 426, then try the "LearnOOP" tutorial in Chapter 6.